Ultracomputer Note #24
TIMING ANALYSIS FOR SCIENTIFIC CODES
RUN UNDER WASHCLOTH SIMULATION

David Korn
28 March 1981

Chapter I

INTRODUCTION

This note details the timing measurements made for scientific programs run under the WASHCLOTH simulator [1,2].

I analyze the processing time as a function of the number of processing elements and the problem size. The aim is predict how the number of processors that can be used efficiently varies as a function of the problem size. In addition, the analysis sheds light on the problem of designing and coding efficient programs.

The results obtained for the reduction of a real symmetric matrix to tridiagonal form using Householder similarity transforms [8] are presented.

CHAPTER 2

BACKGROUND

The model considered here is a Multiple Instruction stream Multiple Data stream (MIMD) machine with #PE identical processing elements (PEs) that share a common or "public" memory. This model was dubbed a paracomputer by Schwartz[7]. The individual PEs may also have attached local memory referred to as "private" memories. The PEs can simultaneously read or write any public location in a single cycle. This model cannot be realized for a large number of processor elements due to fan-in limitations. In order to implement a feasible approximation to a paracomputer, a multicycle interconnection network has been proposed [3] which joins #PE processing elements to #PE memory modules. Synchronization of the processor elements is achieved by adding a machine instruction that has the effect that any number of processing elements may simultaneously increment a given memory location [3]. This instruction is called "replace-add" and can be implemented efficiently within the interconnection network and memory modules [3].

A paracomputer can emulate a vector or single instruction multiple data stream (SIMD) machine by synchronizing after each instruction or vector operation. However, parallelization of computer code is most efficient when preformed at the outermost or highest level. For example, it is preferable to have each PE independently perform a serial matrix diagonalization rather than to have them perform the parallel algorithm discussed in this paper on each of the matrices.

Since parallelization is most effective at the highest level and vector operations most effective at the lowest level we might envision adding additional hardware to each PE to perform low level vectorization. Experience (CRAY) suggests that processing vectors of length < 100 in parallel achieves substantial performance improvement. One could imagine a problem with O(10**7) or more data points being handled by high level parallelization and low level vectorization.

CHAPTER 3

METHODOLOGY

In order to study the performance of scientific programs on a paracomputer, the WASHCLOTH simulator was written [2]. WASHCLOTH simulates a paracomputer consisting of up to approximately 1000 CDC 6600 computers augmented with the replace-add instruction as described above.

The first step in this process is to modify the serial code to conform to WASHCLOTH standards. This involves putting shared variables into public memory (blank common) and establishing working space for WASHCLOTH. The details for this procedure can be found in [1,5]. The number of serial instructions needed to execute the code can then be obtained by running with one PE and 'trapping' a public variable at the beginning and end of a one processor simulated run and subtracting the generated instruction cycle counts.

The next step is to modify the code to permit parallel execution. This is facilitated by several library routines that enable simple conversion of one and two dimensional DO loops for which the loop iterations may be executed in parallel. Then efficient simulation of one processing element is performed using SOFTSOAP [5] and multiple processor simulations are performed using WASHCLOTH. A final test is to use the 'irregular' version of WASHCLOTH to slow down one or more of the processing elements. This often reveals unwarranted assumptions that PEs run at approximately the same rate.

The routines described in [5] print the accumulated waiting time for each processing element. Additional information statistics about public and private memory references is obtained by running with the 'trace' option of WASHCLOTH.

The speedup, S, using #PE processing elements is defined as

$$S = T(1)/T(\#PE)$$

where T(j) is the time required with j processing elements. The efficiency is the speedup divided by #PE. Clearly the speedup lies between 1 and #PE and the efficiency between 0 and 1.

Next simulations are performed varying the problem size and the number of PEs. The data gathered from the experiments indicate how well the problem parallelizes and is also used to

obtain the projection discussed in the next section.

CHAPTER 4

TIMING PROJECTIONS

In this section we describe a technique for projecting the performance of an algorithm for problems and parallel machines too big to simulate. The aim is to predict the performance that would be obtained on a system with any number of processing elements from the data obtained with a few simulations. Toward this end we develop a formula for the required computation time.

It is possible the categorize the number of instructions spent by a processing element into three classes.

The first category contains instructions that are executed by each processor independent of how many processing elements are assigned to solve the problem. This term is often neglected in assymptotic formulas. For example, if the iterations of a DO loop are distributed among several processing elements, it is still necessary to initialize the loop index for each processor. Testing for boundary points is often done by each processing element. In some sense we can view the instructions in this

category as overhead which exists even for serial codes. Similarly, synchronization instructions present overhead needed for parallelization. The exact number of instructions in category one is compiler dependent: Compiler optimizations often increases this number since instructions are added to loop initialization in order to reduce the loop body.

The second category of instructions are those that may be distributed among the PEs. While these instructions may not all be dividable among the processing elements, the total number is independent of the number of processing elements. The bodies of parallelizable loops fall into this category: With more processing elements each processor will execute the loop fewer times. In order to decrease the waiting time (described below) it is desirable to divide these instructions as evenly as possible among the processing elements. The subroutine library described in [5] enable the user to easily perform this division.

The final category of instructions are those spent waiting for other processing elements to reach a synchronization point. This overhead, which depends on the number of PEs, might be minimized by an operating system in which idle PEs are assigned to other jobs. In particular, the average waiting time per processing element for a section of code that uses K processors (K < #PE) is proportional to

$$(\#PE-K)/\#PE$$

The average waiting time includes the sum of these terms.

It is often possible to examine the code and formulate an approximation to this latter component of waiting time. In some problems characterized by a mesh size N the computation can often be split into segments each requiring time which is a power of N. For such problems the only values of K in the formula above are of the form N**J. For some other problems in which all values of K occur, the resulting sum can be approximated by a single term. This phenomenum is illustrated in the next section.

Another contribution to waiting time arises when the amount of work cannot be evenly divided among the processing elements. This happens when the number of loop iterations is not a multiple of the number of processing elements. This waiting time varies irregularly with the number of processing elements and is hard to analyze. Fortunately, for our experiments (using the methodology of [5]) this term has become comparatively small for large problems.

If we denote by TA the number of instructions executed by all processors and denote by TD the number that can be divided among the processors, then the average number of instructions per processing element for a problem of size N, is given by,

$$T(N,\#PE) = TA(N) + TD(N)/\#PE + TW(N,\#PE).$$

where the waiting time function TW has the components described

above.

If the waiting time remains small relative to TA, the number of processor elements that can be assigned to the efficient solution of a given problem is proportional to TD/TA.

The form of the functions TA(N) and TD(N) can often be deduced from the algorithm used. The growth of TW as a function of processing elements can also be estimated analytically. This information together with WASHCLOTH simulation results can be used to project the results for values of N and numbers of PEs too large to simulate.

CHAPTER 5

A TIMING EXPERIMENT

This chapter describes an experiment involving the subroutine TRED2 from the Argonne National Laboratory scientific library EISPACK. This subroutine uses Householder similarity transformations to reduce a real symmetric matrix to tridiagonal form and also accumulate the orthogonal matrix needed for computing the eigenvectors. In this experiment a parallelized version of TRED2 was produced and runs under simulation were analyzed.

Since the complexity of Householder's method grows as N**3 and the rate determining steps were all parallelized, the term TD increases cubically in N. Parts of the algorithm grow quadratically and others grow linearly. The number of loop initializations grows linearly with N and thus the growth of TA is linear.

At each step K of N steps of the algorithm, parallel operations are performed on a square submatrix of order N-K. Thus at most (N-K)**2 processing elements can be used at each step with the remainder idle. From the results of the preceding section we see that at each step, K, for which (N-K)**2 is less than the number of processing elements, there is a term in the waiting function TW proportional to

(#PE-(N-K)**2)/#PE

By examining the detailed code, we see that the coefficient of this term is independent of both K and the problem size N. The sum over K of these terms is proportional to (#PE**1/2)-2. Since the Housholder algorithm requires a constant amount of serial code at each of its N steps, we obtain a waiting term linear in the problem size. Since, the quadratic parts of the serial code are parallelized to only use at most N processing elements, these segments require linear time. The remaining PEs must wait. Thus, when the number of processing elements grows faster than N, most of the PEs wait giving an additional term that grows linear in N.

The final contribution to waiting time results from the fact that the amount of work is not evenly divisibly among the processors at each step: At step K, mod((N-K)**2,#PE) processing elements are idle when (N-K)**2 > #PE. Since there are fewer than N terms in this sum and since the size of each term is

bounded by #PE, this term can not contribute more than linearly with N.

Summarizing these results, we can characterize the expected execution time for TRED2 as

$$T(N,\#PE) = TA(N) + TD(N)/\#PE + TW(N,\#PE)$$

with

$$TA(N) = A*N$$

$$TD(N) = B*N**2 + C*N**3$$

$$TW(N,\#PE) = D*N*(\#PE-1)/\#PE + B*MAX(0,(\#PE-N)/\#PE) + E*(\#PE**1/2 - 2)$$

where A, B, C, D, and E are constants.

The results of runs with one and two processing elements were used to obtain the values for TA and TD and to estimate TW. For a 32 x 32 matrix TA is 26K, TD is 3521K, and TW is small relative to TA. Therefore we expect that about 137 processing elements can be used efficiently. For a 64x64 matrix TA was found to be 52K and TD was 28283K conforming to the expected pattern of growth. In this case about 540 processing elements could be used efficiently.

More detailed results for a 32x32 matrix are presented in Table 1. In addition to the time spent by each processor the average wait time, the speedup, T1/TN, and the efficiency, T1/N*TN, are also listed for runs with various numbers of

processing elements.

Some additional runs were made with different values of N using one and using 32 processing elements. These results are presented in Table 2. Note the linear growth for TA and TW and the cubic growth for T.

The runs for N=32 were used to estimate the coefficients A, B, C, D, and E given above and the resulting formula was used to project the efficiency of this algorithm for different problem sizes and different numbers of processing elements. The results are shown in Table 3. If we make the optimistic assumption that multiprogramming can eliminate all the waiting time the efficiencies rise to the values given in table 4.

REDUCTION TO TRIDIAGONAL FORM

32 x 32

#PE	T	TW	Speedup	Efficiency
1	3547K	0	1.00	1.00
2	1791K	5K	1.98	.99
4	914K	8K	3.88	.97
8	476K	10K	7.45	.93
16	256K	11K	13.82	.86
32	148K	12K	24.04	.75
48	112K	13K	31.63	.66

Table 1

REDUCTION TO TRIDIAGONAL FORM

32 PROCESSING ELEMENTS

N	T (1 PE)	T (32 PE)	TD/TA	SPEEDUP	EFFICIENCY
16	449K	32K	35	13.90	.43
24	1501K	75K	77	20.14	.63
32	3547K	148K	137	24.04	.75
48	11955K	430K	304	27.83	.87
64	28336K	960K	540	29.52	.92

Table 2

REDUCTION TO TRIDIAGONAL FORM

PROJECTIONS OF EFFICIENCIES

N \ #PE	32	128	512	2048	8192
16	.43	.14	.03	.01	.00
32	.76	.41	.14	.03	.01
64	.93	.75	.41	.13	.03
128	.98	.92	.74	.40	.13
256	.99	.98	.92	.74	.40
512	1.00	.99	.98	.92	.74
1024	1.00	1.00	.99	.98	.92

Table 3

REDUCTION TO TRIDIAGONAL FORM

PROJECTIONS OF EFFICIENCIES

WITHOUT WAITING TIME

N \ #PE	32	128	512	2048	8192
16	.54	.22	.07	.02	.00
32	.82	.53	.22	.07	.02
64	.95	.81	.52	.21	.07
128	.99	.95	.81	.52	.21
256	1.00	.99	.95	.81	.52
512	1.00	1.00	.99	.95	.81
1024	1.00	1.00	1.00	.99	.95

Table 4

REFERENCES

[1] Allan Gottlieb, "WASHCLOTH - The Logical Successor to Soapsuds", Ultracomputer Note #12, Courant Institute, N.Y.U., 1980.

[2] Allan Gottlieb, "WASHCLOTH 81", Ultracomputer Note #21, Courant Institute, N.Y.U., 1981.

[3] Allan Gottlieb, B. D. Lubachevsky and Larry Rudolph, "Basic Techniques for the Efficient Coordination of Very Large Numbers of Cooperating Sequential Processors," Ultracomputer Note 16, Courant Institute, N.Y.U., 1980.

[4] D. Heller, "A Survey of Parallel Algorithms in Numerical Linear Algebra," SIAM Review, v.20 pp. 740-777.

[5] David Korn, "Scientific Code Conversion," Ultracomputer Note 23, Courant Institute, N.Y.U., 1981.

[6] J. Schwartz, "Ultracomputers," ACM Transactions on Programming Languages and Systems, Vol. 2, pp. 484-521, 1980.

[7] J. H. Wilkinson, The Algebraic Eigenvalue Problem, Oxford University Press, 1965. 1968.